Deciding Bit-Vector Formulas with mcSAT

Aleksandar Zeljić[1], Christoph M. Wintersteiger[2], and Philipp Rümmer[1]

[1] Uppsala University, Sweden
[2] Microsoft Research

Abstract. The Model-Constructing Satisfiability Calculus (mcSAT) is a recently proposed generalization of propositional DPLL/CDCL for reasoning modulo theories. In contrast to most DPLL(T)-based SMT solvers, which carry out conflict-driven learning only on the propositional level, mcSAT calculi can also synthesise new theory literals during learning, resulting in a simple yet very flexible framework for designing efficient decision procedures. We present an mcSAT calculus for the theory of fixed-size bit-vectors, based on tailor-made conflict-driven learning that exploits both propositional and arithmetic properties of bit-vector operations. Our procedure avoids unnecessary bit-blasting and performs well on problems from domains like software verification, and on constraints over large bit-vectors.

1 Introduction

Fixed-length bit-vectors are one of the most commonly used datatypes in Satisfiability Modulo Theories (SMT), with applications in hardware and software verification, synthesis, scheduling, encoding of combinatorial problems, and many more. Bit-vector solvers are highly efficient, and typically based on some form of SAT encoding, commonly called *bit-blasting,* in combination with sophisticated methods for upfront simplification. Bit-blasting may be implemented with varying degree of laziness, ranging from eager approaches where the whole formula is translated to propositional logic in one step, to solvers that only translate conjunctions of bit-vector literals at a time (for an overview, see [22]). Despite the huge body of research, aspects of bit-vector solving are still considered challenging, including the combination of bit-vectors with other theories (e.g., arrays or uninterpreted functions), large bit-vector problems that are primarily of arithmetic character (in particular when non-linear), and problems involving very long bit-vectors. A common problem encountered in such cases is excessive memory consumption of solvers, especially for solvers that bit-blast eagerly.

A contrived yet instructive example with very long bit-vectors is given in Fig. 1, adapted from a benchmark of the SMT-LIB QF_BV 'pspace' subset [18]. The benchmark tests the overflow behavior of addition. Its model is simple, regardless of the size of bit-vectors x and y (x should consist of only 1-bits, while y should consist of only 0-bits). Finding a model for the formula should in principle be easy, but proves challenging for bit-blasting procedures. Other sources of very long bit-vectors are system memory in hardware verification

```
(set-logic QF_BV)
(declare-fun x () (_ BitVec 29980))
(declare-fun y () (_ BitVec 29980))
(assert (and (bvuge x y) (bvule (bvadd x (_ bv1 29980)) y)))
```

Fig. 1: Simplified example from the 'pspace' subset [18] of SMT-LIB, QF_BV

or heap in software verification (e.g., [4]), chemical reaction networks, or gene regulatory networks (e.g., [35]).

Generally, memory consumption is a limiting factor in the application of bit-vector solvers. Increasing the size of bit-vectors that can efficiently be reasoned about would broaden the range of applications, while also simplifying system models for further analysis. With that in mind, we introduce a new model-constructing decision procedure for bit-vectors that is lazier than previous solvers. The procedure is defined as an instance of the *model-constructing satisfiability calculus* (mcSAT [30]), a framework generalizing DPLL and conflict-driven learning (CDCL) to non-Boolean domains. Like other SMT solvers for bit-vectors, our procedure is defined on top of well-understood SAT technology; unlike most existing solvers, we treat bit-vectors as first-class objects, which enables us to design tailor-made propagation and learning schemes for bit-vector constraints, as well as avoiding bit-blasting of bit-vector operations altogether.

The contributions of this paper are as follows: 1. a novel decision procedure for the theory of bit-vectors that avoids bit-blasting, 2. an extension of the mcSAT calculus to support partial model assignments, 3. a new mcSAT heuristic for generalizing explanations, and 4. an implementation of the procedure and preliminary experimental evaluation of its performance.

1.1 Motivating Examples

We start by illustrating our approach using two examples: a simple bit-vector constraint that illustrates the overall strategy followed by our decision procedure, and a simple family of bit-vector problems on which our procedure outperforms existing bit-blasting-based methods. Consider bit-vectors x, y, z of length 4, and let $\oplus$ denote bit-wise exclusive-or and $\leq_u, <_u$ be unsigned comparison in

$$\phi \quad \equiv \quad x = y + z \;\wedge\; y <_u z \;\wedge\; (\, x \leq_u y + y \;\vee\; x \oplus z = 0001\,)\,.$$

The goal is to find an assignment to x, y, z such that formula evaluates to *true*. Fig. 2 illustrates an application of our algorithm to ϕ (after clausification). Starting from an empty trail, we assert the unit clauses, denoted by implications with empty antecedents (lines 1 and 2 in Fig. 2a). At this point the procedure chooses between making a model assignment to one of the bit-vector variables, or Boolean decisions. Here, we choose to make a *decision* and assume $x \leq_u y + y$ (line 3). Decisions (and model assignments) are denoted with a horizontal line above them in the trail. The Boolean structure of the formula ϕ is now satisfied,

	Trail element
1	$() \to x = y + z$
2	$() \to y <_u z$
3	$x \leq_u y + y$
4	$y \mapsto 1111$
5	$z \mapsto ?$

(a) Infeasible trail

	Trail element
1	$() \to x = y + z$
2	$() \to y <_u z$
3	$x \leq_u y + y$
4	$y <_u z \to \neg(y = 1111)$
5	$y \mapsto 1110$
6	$z \mapsto 1111$
7	$x \mapsto 1101$

(b) Conflicted trail

	Trail element
1	$() \to x = y + z$
2	$() \to y <_u z$
3	$x \leq_u y + y$
4	$y <_u z \to \neg(y = 1111)$
5	$(\ldots) \to \neg(y \geq_u 1000)$
6	$y \mapsto 0111$
7	$z \mapsto 1001$
8	$x \mapsto 0000$

(c) Satisfied trail

Fig. 2: Critical trail states during the execution of our algorithm

so we search for satisfying model assignments to the bit-vector variables. Here, we decide on $y \mapsto 1111$ (line 4 in Fig. 2a). The literal $y <_u z$ now gives a lower bound for z. Our procedure immediately determines that the trail has become *infeasible,* since no value of z will be consistent with $y = 1111$ and $y <_u z$.

We now need an *explanation* to restore the trail to a state where it is not infeasible anymore. In mcSAT, an explanation of a conflict is a valid clause with the property that the trail implies falsity of each of the clause literals. One possible explanation in our case is $\neg(y = 1111) \vee \neg(y <_u z)$. After resolving the explanation against the trail (in reverse order, similar to Boolean conflict resolution in SAT solvers), at the first point where at least one literal in the conflict clause no longer evaluates to *false*, the conflict clause becomes an implication and is put on the trail. In this example, as soon as we undo the assignment $y \mapsto 1111$, the literal $\neg(y = 1111)$ can be assumed (line 4 in Fig. 2b). The procedure makes the next legal assignment $y \mapsto 1110$ (line 5 in Fig. 2b). Bounds propagation using $y <_u z$ then implies the model assignment $z \mapsto 1111$ (line 6 in Fig. 2b). Values of y and z imply a unique value 1101 for x, however, the model assignment $x \mapsto 1101$ is not legal because it violates $x <_u y + y$ when $y = 1110$. By means of bounds propagation we have detected a conflict in $y = 1110 \wedge y <_u z \wedge x = z + y \wedge x <_u y + y$.

Our procedure tries to generalize conflicts, to avoid re-visiting conflicts of similar shape in the future. Generalization is done by weakening the literals of $y = 1110 \wedge y <_u z \wedge x = z + y \wedge x <_u y + y$, and checking if the conflict persists. First, $y = 1110$ is rewritten to $y \leq_u 1110 \wedge y \geq_u 1110$; it is then detected that $y \leq_u 1110$ is redundant, because bounds propagation derives unsatisfiability even without it. Now we weaken the literal $y \geq_u 1110$ by changing the constant, say to $y \geq_u 1000$, and verify using bounds propagation that the conflict persists (Example 3). Weakening $y \geq_u 1000$ further would lead to satisfiability. By negation we obtain a valid explanation $\neg(y \geq_u 1000) \vee \neg(y <_u z) \vee \neg(x = z+y) \vee \neg(x <_u y+y)$, which we use to backtrack the trail to a non-conflicted state (line 5 in Fig. 2c). From this point on straight-forward propagation yields a satisfying solution.

After presenting the basic ideas behind our procedure, we argue that it is well suited to problems that stem from model checking applications. Consider

Algorithm 1: Factorial

```
1 unsigned int factorial = 1u;
2 unsigned int n;
3 for (int i = n; i > 0u; i--) {
4   factorial = factorial * i;
5 }
6 assert ( n <= 1 || f % 2u == 0u)
```

the simple C program shown in Alg. 1. The program computes the factorial of some value n by multiplying the factors starting from n and counting down. We add an assertion at the end, which checks whether `factorial` is even if the value of `n` is greater than one. We use bounded model checking and unwind the loop a fixed number of iterations, to generate formulas of increasing complexity. Fig. 3 shows the performance of mcBV (our prototype) and state-of-the-art solvers on these benchmarks (Boolector [10] is the winner of the QF_BV track of the 2015 SMT competition; Z3 [29] is our baseline as mcBV uses the Z3 parser and preprocessor). On this class of benchmarks, mcBV performs significantly better than Z3 and comparably to Boolector.

1.2 Related Work

The most popular approach to solving bit-vector formulas is to translate them to propositional logic and further into conjunctive normal form via the Tseitin translation [33] (*bit-blasting*), such that an off-the-shelf SAT solver can be used to determine satisfiability. In contrast, our approach does not bit-blast, and we attempt to determine satisfiability directly on the word level. Our technique builds on the Model-Constructing Satisfiability Calculus recently developed by Jovanović and de Moura [30, 23]. Our approach is similar in spirit to previous work by Bardin et al. [1], which avoids bit-blasting by encoding bit-vector problems into integer arithmetic, such that a (customized) CLP solver for finite do-

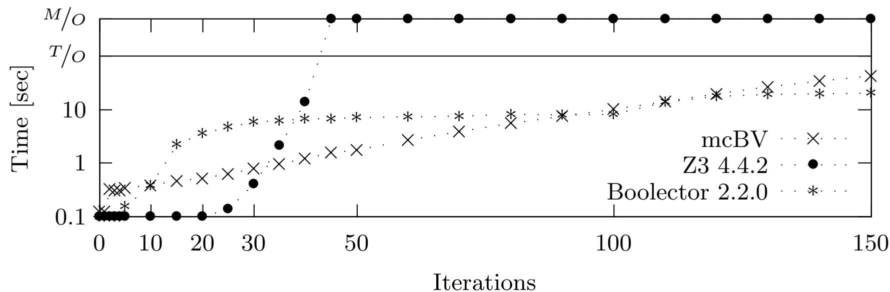


Fig. 3: Factorial example performance

mains can be used. A different angle is taken by Wille et al. [34] in their SWORD tool, which uses vector-level information to increase the performance of the SAT solver by abstracting some sub-formulas into 'modules' that are handled similar to custom propagators in a CLP solver.

On the one hand, various decisions problems involving bit-vectors have recently been shown to be of fairly high complexity [24, 19, 25] and on the other hand, some fragments are known to be decidable in polynomial time; for instance, Bruttomesso and Sharygina describe an efficient decision procedure for the 'core' theory of bit-vectors [11], based on earlier work by Cyrluk et al. [15], who defined this fragment via syntactic restriction to extraction and concatenation being the only bit-vector operators that are permitted. There is also a small body of work on the extension of decision procedures for bit-vectors that do not have a fixed size. For instance, Bjørner and Pichora [7] describe a unification-based calculus for (non-)fixed size bit-vectors, while Möller and Ruess [28] describe a procedure for (non-)fixed size bit-vectors that bit-blasts lazily ('splitting on demand').

Most SMT solvers implement lazy and/or eager bit-blasting procedures. These either directly, eagerly translate to a Boolean CNF and then run a SAT solver, or, especially when theory combination is required, they use a lazy bit-blaster that translates relevant parts of formulas on demand. This is the case, for instance in Boolector [10], MathSAT [13], CVC4 [2], STP [20], Yices [17], and Z3 [29]. Hadarean et al. [22] present a comparison and evaluation of eager, lazy, and combined bit-vector solvers in CVC4.

Griggio proposes an efficient procedure for the construction of Craig interpolants of bit-vector formulas ([27] via translation to QF_LIA, quantifier-free linear integer arithmetic [21]). Interpolants do have applications in mcBV, e.g., for conflict generalization, but we do not currently employ such methods.

Model checkers that do not use SMT solvers sometimes implement their own bit-blasting procedures and then use SAT solvers, BDD-based, or AIG-based solvers. This is often done in bounded model checking [5, 6], but more recently also in IC3 [9], Impact [27], or k-induction [32]. Examples thereof include CBMC [14], EBMC [26], NuSMV2 [12]. In some cases model-checkers based on abstract interpretation principles use bit-vector solvers for counter-example generation when the proof fails; this is the case, for instance, for the separation-logic based memory analyzer SLAyer [3, 4].

In recent times bit-vector constraints are also used for formal systems analysis procedures in areas other than verification, for instance in computational biology [35] where Dunn et al. [16] identify and analyze a bit-vector model for pluripotent stem-cells via an encoding of the model into bit-vector constraints. Similarly, bit-vector solvers are used within interactive and automated theorem provers to construct bit-vector proofs, for instance, in Isabelle/HOL [8].

2 Preliminaries: Bit-Vector Constraints

We consider a logic of quantifier-free fixed-width bit-vector constraints, defined by the following grammar, in which ϕ ranges over formulas and t over bit-vector

terms:

$$\begin{aligned} \phi &::= \ \mathit{true} \mid \mathit{false} \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid t \bullet t \\ t &::= \ 0_n \mid 1_n \mid \cdots \mid (0|1)^+ \mid x \mid \mathsf{extract}^n_p(t) \mid \,!\,t \mid t \circ t \end{aligned}$$

Here, p ranges over propositional variables; expressions $0_n, 1_n, \ldots$ are decimal bit-vector literals of width n; literals $(0|1)^+$ represent bit-vectors in binary; x ranges over bit-vector variables of size $\alpha(x)$; predicates $\bullet \in \{=, \leq_s, \leq_u\}$ represent equality, signed inequality (2's complement format), and unsigned inequality, respectively; the operator $\mathsf{extract}^n_p$ represents extraction of n bits starting from position p (where the left-most bit has position 0); ! is bit-wise negation; binary operators $\circ \in \{+, \times, \div, \&, |, \oplus, \ll, \gg, \frown\}$ represent addition, multiplication, (unsigned) integer division, bit-wise and, bit-wise or, bit-wise exclusive or, left-shift, right-shift, and concatenation, respectively. We assume typing and semantics of bit-vector constraints are defined as usual.

An *atom* is a formula ϕ that does not contain $\neg, \wedge, \vee$. An atom is *flat* if it is of the form $x \leq_s y$, $x \leq_u y$, or $x = t$, and t does not contain nested operators. A *literal* is an atom or its negation. A *clause* is a disjunction of literals. When checking satisfiability of a bit-vector constraint, we generally assume that the constraint is given in the form of a set of clauses containing only flat atoms.

3 mcSAT with Projections

We now introduce the framework used by our decision procedure for bit-vector constraints, based on a generalized version of mcSAT [30]. In contrast to previous formulations of mcSAT, we include the possibility to *partially assign* values to variables; this enables assignments that only affect some of the bits in a bit-vector, which helps us to define more flexible propagation operators. mcSAT with projections is first defined in general terms, and tailored to the setting of bit-vector constraints in the subsequent sections, resulting in mcBV.

We define our framework in the form of a transition system, following the tradition of DPLL(T) [31] and mcSAT [30]. The states of the system have the form $\langle M, C \rangle$, where M is the *trail* (a finite sequence of *trail elements*), and C is a set of clauses. The trail M consists of: (1) *decided literals* l, (2) *propagated literals* $e \to l$, and (3) *model assignments* $\pi(x) \mapsto \alpha$. A literal l (either decided or propagated) is considered to be true in the current state if it appears in M, which is denoted by $l \in M$. Model assignments $\pi(x) \mapsto \alpha$ denote a *partial assignment* of a value α to a variable x, where π is a *projection function*.

We consider constraints formulated over a set of types $\{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_n\}$ with fixed domains $\{T_1, T_2, \ldots, T_n\}$, and a finite family $\{\pi_i\}_{i \in I}$ of surjective functions $\pi_i : T \mapsto T'$ (here called projections) between the domains. Types can for instance be bit-vector sorts of various lengths. A partial model assignment $\pi(x) \mapsto \alpha$ with projection $\pi : T \mapsto T'$ expresses that variable x of type $\mathcal{T}$ is assigned some value $\beta \in T$ such that $\pi(\beta) = \alpha$, where $\alpha \in T'$. A trail can contain multiple partial assignments to the same variable x; we define the *partial domain* of a

variable x under a trail M as

$$Domain(x, M) = \bigcap_{(\pi(x) \mapsto \alpha) \in M} \{\beta \in T \mid \pi(\beta) = \alpha\} \ .$$

We call a trail M *assignment consistent* if the partial assignments to variables are mutually consistent, i.e., the partial domain $Domain(x, M)$ of each variable x is non-empty. If the partial domain of a variable x contains exactly one element, i.e., $Domain(x, M) = \{\beta\}$, then we say that all the partial assignments to x in M form a *full model assignment*; in the original mcSAT calculus, this is denoted by $x \mapsto \beta$. Assignment consistency is violated if $Domain(x, M) = \varnothing$. We generally require that projections $\{\pi_i\}_{i \in I}$ are chosen in such a way that assignment consistency and full assignments can be detected effectively for any trail M. In addition, projections are required to be *complete* in the sense that every full model assignment $x \mapsto \beta$ can be expressed as some finite combination of partial assignments $\{\pi_j(x) \mapsto \alpha_j\}_{j \in J}$. More formally, for every $\beta \in T$ there exists a finite set of partial model assignments S such that $Domain(x, S) = \{\beta\}$. Inclusion of the *identity function* among projections enables expression of full model assignments directly.

Given a trail M, an interpretation $v[M] = \{x_1 \mapsto \beta_1, x_2 \mapsto \beta_2, \ldots, x_k \mapsto \beta_k\}$ is constructed by collecting all full model assignments $x_i \mapsto \beta_i$ implied by M. The value $v[M](t)$ of a term or formula t is its value under the interpretation v, provided that all variables occurring in t are interpreted by v; or *undef* otherwise. We define a *trail extension* $\bar{M}$ of a trail M as any trail $\bar{M} = [M, M']$ such that M' consists only of (partial) model assignments to variables already appearing in M, and furthermore each variable x that appears in M has a unique value assigned from its partial domain; $Domain(x, M) \neq \emptyset$ implies that $|Domain(x, \bar{M})| = 1$. This ensures that assignment consistency is maintained.

Evaluation of literals in respect to the trail M is achieved using a pair of functions $value_B$ and $value_T$, defined as

$$value_B(l, M) = \begin{cases} true & l \in M \\ false & \neg l \in M \\ undef & \text{otherwise} \end{cases} \quad \text{and} \quad value_T(l, M) = v[M](l) \ .$$

A trail M is *consistent* if it is assignment consistent, and for all literals $l \in M$ it holds that $value_T(l, M) \neq false$. A trail M is said to be *complete* if it is consistent and every literal l on the trail M can be evaluated in the theory, i.e. $value_T(l, M) = true$. A trail which has no complete extensions is called *infeasible*. Note that if a trail is inconsistent then it is also infeasible.

The value of a literal in a consistent state (consistent trail) is defined as

$$value(l, M) = \begin{cases} value_B(l, M) & value_B(l, M) \neq undef \\ value_T(l, M) & \text{otherwise} \end{cases} ,$$

which is extended to clauses in the obvious way.

T-Decide

$$\langle M, C\rangle \longrightarrow \langle [M, \pi(x) \mapsto \alpha], C\rangle \quad \textbf{if} \quad \begin{array}{l} x \text{ is a (theory) variable in } C \\ \mathit{Domain}(x, [M, \pi(x) \mapsto \alpha]) \neq \mathit{Domain}(x, M) \\ [M, \pi(x) \mapsto \alpha] \text{ is consistent} \end{array}$$

Fig. 4: The modified T-Decide rule.

Evaluation strength. We remark that there is some freedom in the way value_T is defined: even if $v[M](l) = \mathit{undef}$ for some literal l (because l contains variables with undefined value), the trail M might still uniquely determine the value of l. In general, our calculus can use any definition of value^*_T that satisfies

(1) $v[M](l) \neq \mathit{undef}$ implies $\mathit{value}^*_T(l, M) = v[M](l)$, and
(2) $\mathit{value}^*_T(l, M) \neq \mathit{undef}$ implies that for every extension $\bar{M}$ of M it holds that $\mathit{value}_T(l, \bar{M}) = \mathit{value}^*_T(l, M)$.

These properties leave room for a trade-off between the strength of reasoning and computational effort invested to discover such implications. For example, suppose that a bit-vector variable x of length 3, under trail M has the partial domain $\mathit{Domain}(x, M) = \{000, 001, 010\}$. For a literal $l = (x < 100)$, evaluation yields $\mathit{value}_T(l, M) = \mathit{undef}$. It is easy to see that $\mathit{value}_T(l, \bar{M}) = \mathit{true}$ in every trail extension $\bar{M}$ of M, however, so that a lazier mode of evaluation could determine $\mathit{value}^*_T(l, M) = \mathit{true}$. With a more liberal evaluation strategy, propagations and conflicts are detected earlier, though perhaps at higher cost.

3.1 A Calculus with Projections

The transitions of our calculus are the same as those of mcSAT [30], with the exception of the T-Decide rule, which we define in terms of partial assignments and partial domains (Fig. 4). As in mcSAT, it is assumed that a *finite basis* $\boldsymbol{B}$ of literals is given, representing all literals that are taken into account in decisions, propagations, or when constructing conflict clauses and explanations. $\boldsymbol{B}$ at least has to contain all atoms, and the negation of atoms occurring in the clause set C. The function *explain* is supposed to compute *explanations* of infeasible trails M (which correspond to theory lemmas in DPLL(T) terminology). An explanation of M is a clause e such that 1. e is valid; 2. all literals $l \in e$ evaluate to false on M (i.e., $\mathit{value}(l, M) = \mathit{false}$); 3. all literals $l \in e$ occur in the basis $\boldsymbol{B}$.

In order to state correctness of the calculus, we need one further assumption about the well-foundedness of partial assignments: for every sequence of partial assignments to a variable x, of the form $[\pi_1(x) \mapsto \alpha_1, \pi_2(x) \mapsto \alpha_2, \ldots]$, we assume that the sequence of partial prefix domains

$$\begin{array}{l} \mathit{Domain}(x, []) \\ \mathit{Domain}(x, [\pi_1(x) \mapsto \alpha_1]) \\ \mathit{Domain}(x, [\pi_1(x) \mapsto \alpha_1, \pi_2(x) \mapsto \alpha_2]) \\ \quad \ldots \end{array}$$

eventually becomes constant. This ensures that partial assignment of a variable cannot be refined indefinitely. Correctness of mcSAT with projections is then be proven in largely the same manner as in mcSAT:

Theorem 1 (Correctness [30]). *Any derivation starting from the initial state $\langle [], C \rangle$ eventually terminates in state sat, if C is satisfiable, or in state unsat, if C is unsatisfiable.*

4 Searching for Models with mcBV

We now describe how the mcSAT calculus with projections is tailored to the theory of bit-vectors, leading to our procedure mcBV. The theory of bit-vectors already contains a natural choice for the projections, namely the extract functions, of which we use a finite subset as projections. To ensure *completeness* of this subset (in the sense that every full model assignment has a representation as a combination of partial model assignments), we include all one-bit projections $\pi_i^k = \mathsf{extract}_i^1$, selecting the i-th bit of a bit-vector of length k.

In practice, our prototype implementation maintains a trail M as part of its state, and attempts to extend the trail with literals and model assignments such that the trail stays consistent, every literal on the trail eventually becomes justified by a model assignment (i.e., $value_T(l, M) = true$ for every literal l in M), and every clause in C is eventually satisfied. A conflict is detected if either some clause in C is found to be falsified by the chosen trail elements (which is due to literals or model assignments), or if infeasibility of the trail is detected.

Since the calculus is *model constructing,* there is a strong preference to justify all literals on the trail through model assignments, i.e., to make the trail complete, before making further Boolean decisions. Partial model assignments are instrumental for this strategy: they enable flexible implementation of propagation rules that extract as much information from trail literals as possible. For instance, if the trail contains the equation $x = \mathsf{extract}_2^3(y)$ and a model assignment $x \mapsto 101$, propagation infers and puts a further partial assignment $\mathsf{extract}_2^3(y) \mapsto 101$ on the trail, by means of the T-Decide rule. This partial assignment is subsequently used to derive further information about other variables. For this, we defined bit-precise propagation rules for all operators; our solver includes native implementations of those rules and does not have to resort to explicit bit-blasting. Similarly to Boolean Constraint Propagation (BCP), propagation on the level of bit-vectors is often able to detect inconsistency of trails (in particular variables with empty partial domain) very efficiently.

Once all possible bit-vector propagations have been carried out, but the trail M is still not complete, the values of further variables have to be decided upon through T-Decide. In order to avoid wrong decisions and obvious conflicts, our implementation also maintains over-approximations of the set of feasible values of each variable, in the form of *bit-patterns* and *arithmetic intervals.* These sets are updated whenever new elements are pushed on the trail, and refined using BCP-equivalent propagation and interval constraint propagation (ICP). Besides indicating values of variables consistent with the trail, these sets offer

cheap infeasibility detection (when they become empty), which is crucial for the T-Propagate and T-Conflict rules. Also, frequently one of these sets becomes singleton, in which case a unique model assignment for the variable is implied.

4.1 Efficient Representation of Partial Model Assignments

Our procedure efficiently maintains information about the partial domains of variables by tracking *bit-patterns,* which are strings over the 3-letter alphabet $\{0, 1, u\}$; the symbol u represents undefined bits (*don't-cares*). We say that bit-vector x *matches* bit-pattern p (both of length k) iff x is included in the set of vectors covered in the bit-pattern; formally we define

$$matches(x, p) = \bigwedge_{\substack{0 \le i < k \\ p_i \ne u}} x_i = p_i \ ,$$

where x_i and p_i denote i-th bit of x and p. The atom $matches(x, p)$ is not a formula in the sense of our language of bit-vector constraints, but for the sake of presentation we treat it as such in this section.

For long bit-vectors, representation of partial domains using simple bit-patterns can be inefficient, since linear space is needed in the length of the bit-vector variable. To offset this, we use *run-length encoding (RLE)* to store bit-patterns. Besides memory compression, RLE speeds up propagation, as it is not necessary to process every individual bit of a bit-vector separately. The complexity then depends on the number of bit alternations, as shown in the following example demonstrating exclusive-or evaluation on both representations.

Example 1. Each digit in the output represents one bit operation, in standard bit-vectors (left) and run-length encoded bit-vectors (right) :

x	0000011111		x	$0^3\ 0^2\ 1^3\ 1^2$
y	1110000011		y	$1^3\ 0^2\ 0^3\ 1^2$
x ⊕ y	1110011100		x ⊕ y	$1^3\ 0^2\ 1^3\ 0^2$

4.2 Maintaining Partial Domain Over-approximations

To capture arithmetic properties and enable efficient propagation, our implementation stores bounds $x \in [x_l, x_u]$ for each variable x. The bounds are updated when new elements occur on the trail, and bounds propagation is used to refine bounds. Note that bit-patterns and arithmetic bounds abstractions sometimes also refine each other. For example, lower and upper bounds are derived from a bit-pattern, by replacing all u bits with 0 and 1 respectively. Conversely, if the lower and upper bound share a prefix, then they imply a bit-pattern with the same prefix and the remaining bits set to u.

5 Conflicts and Explanations

Explanations are the vehicle used by our calculus to generalize from conflicts. Given an infeasible trail M, an explanation $\mathit{explain}(M)$ is defined to be a valid clause $E = l_1 \vee \cdots \vee l_n$ over the finite basis $\boldsymbol{B}$, such that every literal l_i evaluates to *false* under the current trail, i.e., $\mathit{value}(l_i, M) = \mathit{false}$ for every $i \in \{1, \ldots, n\}$. Explanations encode contradictory assumptions made on the trail, and are needed in the T-Consume and T-Conflict rules to control conflict resolution and backtracking, as well as in T-Propagate to justify literals added to the trail as the result of theory propagation.

Since the trail M is inconsistent at the beginning of conflict resolution, it is always possible to find explanations that are simply disjunctions of negated trail literals; to this end, every propagated literal $c \to l$ is identified with l, and every model assignment $\pi(x) \mapsto \alpha$ as the formula $\pi(x) = \alpha$.

5.1 Greedy generalization

We present a greedy algorithm for creating explanations that abstract from concrete causes of conflict. To this end, we assume that we have already derived some correct (but not very general) explanation

$$e \;=\; \neg t_1 \vee \neg t_2 \vee \cdots \vee \neg t_n \vee \neg b_1 \vee \neg b_2 \vee \cdots \vee \neg b_m \;,$$

where $t_1, \ldots, t_n$ denote literals with $\mathit{value}_T(t_i, M) = \mathit{true}$, and $b_1, \ldots, b_m$ literals with $\mathit{value}_T(t_i, M) = \mathit{undef}$ but $\mathit{value}_B(b_i, M) = \mathit{true}$. The former kind of literal holds as a result of model assignments, whereas the latter literals occur on the trail either as decisions or as the result of propagation. The key observation is that the literals $t_1, \ldots, t_n$ allow over-approximations (replacements with logically weaker literals), as long as the validity of the overall explanation clause is maintained, in this way producing a more general explanation.

Our procedure requires the following components as input (apart from e):

- for each literal t_i (for $i \in \{1, \ldots, n\}$), a finite lattice $(T_i, \Rightarrow)$ of conjunctions of literals $T_i \subseteq \{l_1 \wedge \cdots \wedge l_k \mid l_1, \ldots, l_k \in \boldsymbol{B}\}$ ordered by logical implication, with join $\sqcup_i$ and meet $\sqcap_i$, and the property that $t_i \in T_i$. The set T_i provides constraints that are considered as relaxation of t_i.
- a heuristic satisfiability checker *hsat* to determine the satisfiability of a conjunction of literals. The checker *hsat* is required to (1) be be sound (i.e., $\mathit{hsat}(\phi) = \mathit{false}$ implies that ϕ is actually unsatisfiable), (2) to correctly report the validity of e,

  $$\mathit{hsat}(t_1 \wedge \cdots \wedge t_n \wedge b_1 \wedge \cdots \wedge b_m) = \mathit{false},$$

  and (3) to be *monotonic* in the following sense: for all elements $l, l' \in T_i$ with $l \Rightarrow l'$ in one of the lattices, and for all conjunctions ϕ, ψ of literals, if $\mathit{hsat}(\phi \wedge l' \wedge \psi) = \mathit{false}$ then $\mathit{hsat}(\phi \wedge l \wedge \psi) = \mathit{false}$.

Algorithm 2: Explanation relaxation

Input: Raw explanation $\bigvee_{i=1}^{n} \neg t_i \vee \bigvee_{i=1}^{m} \neg b_i$;
lattices $(T_i, \Rightarrow)_{i=1}^{n}$; satisfiability checker *hsat*.
Output: Refined explanation $\bigvee_{i=1}^{n} \neg t_i^a \vee \bigvee_{i=1}^{m} \neg b_i$.

```
1  φ_b ← b_1 ∧ ··· ∧ b_m;
2  for i ← 1 to n do
3  |   t_i^a ← t_i;
4  |   B_i ← ∅;
5  end
6  changed ← true;
7  while changed do
8  |   changed ← false;
9  |   for i ← 1 to n do
10 |   |   if ∃t ∈ T_i \ {t_i^a} with t_i^a ⇒ t and ∀l ∈ B_i. t ⇏ l then
11 |   |   |   if hsat(⋀_{j=1}^{i-1} t_j^a ∧ t ∧ ⋀_{j=i+1}^{n} t_j^a ∧ φ_b) then
12 |   |   |   |   B_i ← B_i ∪ {t};
13 |   |   |   else
14 |   |   |   |   t_i^a ← t;
15 |   |   |   |   changed ← true;
16 |   |   |   end
17 |   |   end
18 |   end
19 end
20 return ¬t_1^a ∨ ··· ∨ ¬t_n^a ∨ ¬b_1 ∨ ··· ∨ ¬b_m;
```

The pseudo-code of the procedure is shown in Alg. 2, and consists mainly of a fixed-point loop in which the literals $t_1, \ldots, t_n$ are iteratively weakened, until no further changes are possible (lines 6–19). The algorithm keeps blocking sets B_i of conjunctions of literals (for $i \in \{1, \ldots, n\}$) that have been considered as relaxation for t_i, but were found to be too weak to maintain a valid explanation.

Lemma 1. *Provided a correct explanation clause as input, Alg. 2 terminates and produces a correct refined explanation.*

The next two sections describe two instances of our procedure: one targeting explanations that are primarily of arithmetic character, and one for explanations that mainly involve bit-wise operations.

5.2 Greedy Bit-wise Generalization

If the literals of an explanation clause are primarily bit-wise in nature, the relaxation considered in our method is to weaken the bit-patterns associated with the variables occurring in the conflict. For every literal t_i of the form $matches(x, p)$,

Clause	Result
$\neg matches(y,\ 0000) \vee \neg matches(x,\ 10) \vee \neg \mathsf{extract}_2^2(y) = x$	Valid
$\neg matches(y,\ \boldsymbol{u}000) \vee \neg matches(x,\ 10) \vee \neg \mathsf{extract}_2^2(y) = x$	Valid
$\neg matches(y,\ uu\boldsymbol{u}0) \vee \neg matches(x,\ 10) \vee \neg \mathsf{extract}_2^2(y) = x$	$y = 0010, x = 10$
$\neg matches(y,\ uu00) \vee \neg matches(x,\ 10) \vee \neg \mathsf{extract}_2^2(y) = x$	Valid
$\neg matches(y,\ uu0\boldsymbol{u}) \vee \neg matches(x,\ 10) \vee \neg \mathsf{extract}_2^2(y) = x$	Valid
$\neg matches(y,\ uu0u) \vee \neg matches(x,\ \boldsymbol{u}0) \vee \neg \mathsf{extract}_2^2(y) = x$	$y = 0000, x = 00$
$\neg matches(y,\ uu0u) \vee \neg matches(x,\ 1\boldsymbol{u}) \vee \neg \mathsf{extract}_2^2(y) = x$	Valid

Fig. 5: Generalization based on bit-patterns

where x is a bit-vector variable and p a bit-pattern of width $k = \alpha(x)$ implied by the trail, we choose the lattice $(T_i, \Rightarrow)$ with

$$T_i \ = \ \{\mathit{false}\} \cup \{matches(x, a) \mid a \in \{0, 1, u\}^k\} \ .$$

This set contains a constraint that is equivalent to *true*, namely the literal $matches(x, u^k)$. Conflicts involving (partial) assignments allow us to start near the bottom of the lattice and we weaken the literal as much as possible in order to cover as many similar assignments as possible. Concretely, the weakening is performed by replacing occurrences of 1 or 0 in the bit-pattern by u.

Our prototype satisfiability checker *hsat* for this type of constraints is implemented by propagation (Sect. 4), which is able to show validity of raw explanation clauses, and similarly handles bit-pattern relaxations. Our implementation covers all operations, but it is imprecise for some arithmetic operations.

Example 2. Consider the trail

$$M = [\ldots, y \mapsto 0^4, x \mapsto 1^1 0^1, \mathsf{extract}_2^2(y) = x] \ .$$

This trail is inconsistent because literal $\mathsf{extract}_2^2(y) = x$ evaluates to false in the theory. A simple explanation clause is $\neg matches(y, 0^4) \vee \neg matches(x, 1^1 0^1) \vee \neg(\mathsf{extract}_2^2(y) = x)$. The generalization procedure now tries to generalize this naive explanation by weakening the literals. Fig. 5 shows steps of generalizing the conflict observed in trail M. One by one, bits in the pattern are set to u and it is checked whether the new clause is valid. If it is valid, the new bit-pattern is altered further, otherwise we discard it and continue with the last successfully weakened pattern. Note that we are not restricted to changes to only one bit, or even only one literal at a time.

5.3 Greedy Arithmetic Generalization

If the literals of an explanation clause are primarily arithmetic, the relaxation considered in our method is to replace equations (that stem from model assignments on the trail) with inequalities or interval constraints: for every literal t_i of the form $x = v_k$, where x is a bit-vector variable and v_k a literal bit-vector constant of width $k = \alpha(x)$, we choose the lattice $(T_i, \Rightarrow)$ with

$$\begin{aligned} T_i \ = \ & \{\mathit{false}\} \cup \{x = a \mid a \in \{0,1\}^k\} \cup \\ & \{a \leq_u x \wedge x \leq_u b \mid a, b \in \{0,1\}^k, a < b\} \end{aligned}$$

$\neg(y \leq_u 14) \vee \neg(y \geq_u 14) \vee \neg(y <_u z) \vee \neg(x = z + y) \vee \neg(x <_u y + y)$	Valid
$\neg(y \leq_u 15) \vee \neg(y \geq_u 14) \vee \neg(y <_u z) \vee \neg(x = z + y) \vee \neg(x <_u y + y)$	Valid
$\neg(y \geq_u 0) \vee \neg(y <_u z) \vee \neg(x = z + y) \vee \neg(x <_u y + y)$	$y = 1, \ldots$
$\neg(y \geq_u 7) \vee \neg(y <_u z) \vee \neg(x = z + y) \vee \neg(x <_u y + y)$	$y = 7, \ldots$
$\neg(y \geq_u 10) \vee \neg(y <_u z) \vee \neg(x = z + y) \vee \neg(x <_u y + y)$	Valid
$\neg(y \geq_u 8) \vee \neg(y <_u z) \vee \neg(x = z + y) \vee \neg(x <_u y + y)$	Valid

Fig. 6: Generalization based on arithmetic and interval bounds propagation

This set contains a constraint that is equivalent to *true* ($0^k \leq_u x \wedge x \leq_u 1^k$), and similarly constraints that only impose concrete lower or upper bounds on x, and equalities $(x = v_k) \in T_i$.

Our satisfiability checker *hsat* for this type of constraints implements interval constraint propagation, covering all bit-vector operations, although it tends to yield more precise results for arithmetic than for bit-wise operations. If ICP shows that an interval becomes empty, then the generalization succeeds because the conflict persists. Otherwise, generalization fails when ICP reaches a fix point or exceeds a fixed number of steps (to avoid problems with slow convergence).

Example 3. For readability purposes we switch to numerical notation in this example. Recall the basic explanation of the trail conflict shown in Fig. 2b in the motivating example (Sect. 1.1):

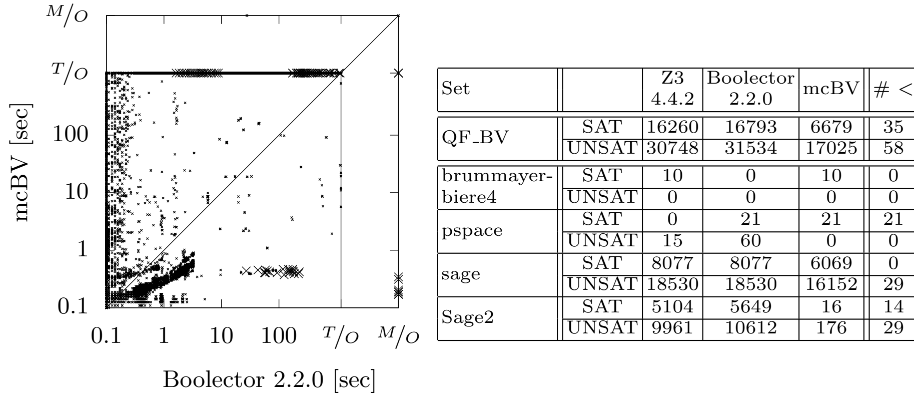
$$\neg(y = 14) \vee \neg(y <_u z) \vee \neg(x = z + y) \vee (x <_u y + y)$$

We rewrite the first negated equality as a disjunction of negated inequalities. Then the iterative generalization procedure starts. For each bound literal, the procedure first attempts to remove it (by weakening it to a *true*-equivalent in the lattice T_i). If unsuccessful, it navigates the lattice of literals using binary search over bounds. Fig. 6 shows the steps in this particular example.

6 Experiments and Evaluation

To evaluate the performance of our mcBV implementation we conducted experiments on the SMT-LIB QF_BV benchmark set, using our implementation of mcBV in F#, on a Windows HPC cluster of Intel Xeon machines. The benchmark set contains 49971 files in SMT2 format, each of which contains a set of assertions and a single (check-sat) command. The timeout for all experiments is at 1200 sec and the memory limit is 2 GB.

Currently we do not implement any advanced heuristics for clause learning, clause deletion, or restarts and thus mcBV does not outperform any other solver consistently. We present a runtime comparison of mcBV with the state-of-the-art SMT solvers Boolector and Z3 on a selected subset of the QF_BV benchmarks see Fig. 7. On the whole benchmark set, mcBV is not yet competitive with Boolector or Z3, but it is interesting to note that mcBV performs well on some of the benchmarks in the 'sage', 'sage2' and 'pspace' sets, as well as the entirety of the



Set		Z3 4.4.2	Boolector 2.2.0	mcBV	# <
QF_BV	SAT	16260	16793	6679	35
	UNSAT	30748	31534	17025	58
brummayer-biere4	SAT	10	0	10	0
	UNSAT	0	0	0	0
pspace	SAT	0	21	21	21
	UNSAT	15	60	0	0
sage	SAT	8077	8077	6069	0
	UNSAT	18530	18530	16152	29
Sage2	SAT	5104	5649	16	14
	UNSAT	9961	10612	176	29

Fig. 7: Runtime comparison on selected subsets of SMT QF_BV. Markers for 'sage' and 'sage2' are smaller to avoid clutter; #< shows the number of benchmarks that only mcBV solves or mcBV solves quicker than both Z3 and Boolector.

'brummayerbiere4' set. Those sets contain a substantial number of benchmarks that mcBV could solve, but Z3 and Boolector cannot. The 'pspace' benchmarks are hard for all solvers as they contains very large bit-vectors (in the order of 20k bits) which will often result in the bit-blaster running out of memory; this is reflected in the small clusters at the bottom right in Fig. 7. The table in Fig. 7 gives the number of instances solved by each approach. While our prototype performs relatively well on selected subsets, it will need improvements and advanced heuristics to compete with the state-of-the-art on all of QF_BV.

7 Conclusion

We presented a new decision procedure of the theory of bit-vectors, which is based on an extension of the Model-Constructing Satisfiability Calculus (mcSAT). In contrast to state-of-the-art solvers, our procedure avoids unnecessary bit-blasting. Although our implementation is prototypical and lacks most of the more advanced heuristics used in solvers (e.g., variable selection/decision heuristics, lemma learning, restarts, deletion strategies), our approach shows promising performance, and is comparable with the best available solvers on a number of benchmarks. This constitutes a proof of concept for instantiation of the mcSAT framework for a new theory; we expect significantly improved performance as we further optimise our implementation. Additionally, we improve the flexibility of the mcSAT framework by introducing projection functions and partial assignments, which we believe to be crucial for the model-constructing approach for bit-vectors.

References

1. Bardin, S., Herrmann, P., Perroud, F.: An alternative to SAT-based approaches for bit-vectors. In: TACAS. LNCS, vol. 6015. Springer (2010)
2. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Proc. CAV. LNCS, vol. 6806. Springer (2011)
3. Berdine, J., Cook, B., Ishtiaq, S.: SLAyer: Memory safety for systems-level code. In: CAV. LNCS, vol. 6806. Springer (2011)
4. Berdine, J., Cox, A., Ishtiaq, S., Wintersteiger, C.M.: Diagnosing abstraction failure for separation logic-based analyses. In: CAV. LNCS, vol. 7358. Springer (2012)
5. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic model checking using SAT procedures instead of BDDs. In: DAC (1999)
6. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: TACAS. LNCS, vol. 1579. Springer (1999)
7. Bjørner, N., Pichora, M.C.: Deciding fixed and non-fixed size bit-vectors. In: TACAS. LNCS, vol. 1384. Springer (1998)
8. Böhme, S., Fox, A.C.J., Sewell, T., Weber, T.: Reconstruction of Z3's bit-vector proofs in HOL4 and Isabelle/HOL. In: Certified Programs and Proofs (CPP). LNCS, vol. 7086. Springer (2011)
9. Bradley, A.R.: SAT-based model checking without unrolling. In: VMCAI. LNCS, vol. 6538. Springer (2011)
10. Brummayer, R., Biere, A.: Boolector: An efficient SMT solver for bit-vectors and arrays. In: TACAS. LNCS, vol. 5505. Springer (2009)
11. Bruttomesso, R., Sharygina, N.: A scalable decision procedure for fixed-width bit-vectors. In: ICCAD. ACM (2009)
12. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In: CAV. LNCS, vol. 2404. Springer (2002)
13. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: TACAS. LNCS, vol. 7795. Springer (2013)
14. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. LNCS, vol. 2988. Springer (2004)
15. Cyrluk, D., Möller, M.O., Rueß, H.: An efficient decision procedure for the theory of fixed-sized bit-vectors. In: CAV. LNCS, vol. 1254. Springer (1997)
16. Dunn, S.J., Martello, G., Yordanov, B., Emmott, S., Smith, A.: Defining an essential transcriptional factor program for naïve pluripotency. Science 344(6188) (2014)
17. Dutertre, B.: System description: Yices 1.0.10. In: SMT-COMP'07 (2007)
18. Froehlich, A., Kovasznai, G., Biere, A.: Efficiently solving bit-vector problems using model checkers. In: SMT Workshop (2013)
19. Fröhlich, A., Kovásznai, G., Biere, A.: More on the complexity of quantifier-free fixed-size bit-vector logics with binary encoding. In: Symp. Comp. Sci. - Theory and Applications (CSR). LNCS, vol. 7913. Springer (2013)
20. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: CAV. LNCS, vol. 4590. Springer (2007)
21. Griggio, A.: Effective word-level interpolation for software verification. In: FMCAD. FMCAD Inc. (2011)
22. Hadarean, L., Bansal, K., Jovanovic, D., Barrett, C., Tinelli, C.: A tale of two solvers: Eager and lazy approaches to bit-vectors. In: CAV. LNCS, vol. 8559. Springer (2014)

23. Jovanovic, D., de Moura, L.M.: Cutting to the chase - solving linear integer arithmetic. J. Autom. Reasoning 51(1) (2013)
24. Kovásznai, G., Fröhlich, A., Biere, A.: On the complexity of fixed-size bit-vector logics with binary encoded bit-width. In: SMT. EPiC Series, vol. 20. EasyChair (2013)
25. Kovásznai, G., Veith, H., Fröhlich, A., Biere, A.: On the complexity of symbolic verification and decision problems in bit-vector logic. In: Mathematical Foundations of Comp. Sci. MFCS. LNCS, vol. 8635. Springer (2014)
26. Kroening, D.: Computing over-approximations with bounded model checking. In: BMC Workshop. vol. 144 (January 2006)
27. McMillan, K.L.: Interpolation and SAT-based model checking. In: CAV. LNCS, vol. 2725. Springer (2003)
28. Möller, M.O., Rueß, H.: Solving bit-vector equations. In: FMCAD. LNCS, vol. 1522. Springer (1998)
29. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. LNCS, vol. 4963. Springer (2008)
30. de Moura, L., Jovanovic, D.: A model-constructing satisfiability calculus. In: VMCAI (2013)
31. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). J. ACM 53(6) (2006)
32. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: FMCAD. LNCS, vol. 1954. Springer (2000)
33. Tseitin, G.: On the complexity of derivation in propositional calculus. Studies in Constructive Mathematics and Mathematical Logic, Part II, Seminars in Mathematics (1970), translated from Russian: Zapiski Nauchnykh Seminarov LOMI 8 (1968)
34. Wille, R., Fey, G., Große, D., Eggersglüß, S., Drechsler, R.: SWORD: A SAT like prover using word level information. In: Intl. Conf. on Very Large Scale Integration of System-on-Chip (VLSI-SoC 2007). IEEE (2007)
35. Yordanov, B., Wintersteiger, C.M., Hamadi, Y., Kugler, H.: SMT-based analysis of biological computation. In: NASA Formal Methods NFM. LNCS, vol. 7871. Springer (2013)